Chapter 4. The Publisher Web Service

The Publisher web service is a demonstration of a more complex web service modeled after the one used by the SOAP Web Services Resource Center (http://www.soap-wrc.com/). This service demonstrates techniques for implementing more complicated forms of web services. It builds on the Hello World example from Chapter 3.

4.1 Overview

The Publisher web service manages a database of important news items, articles, and resources that relate to SOAP and web services in general.

A Perl-based service allows registered users to post, delete, or browse items, and to manage their registration information. We've also implemented an interactive Java shell client that uses the Apache SOAP client.

The supported operations are:

register

Create a new user account.

modify

Modify a user account.

login

Start a user session.

post

Post a new item to the database.

remove

Remove an item from the database.

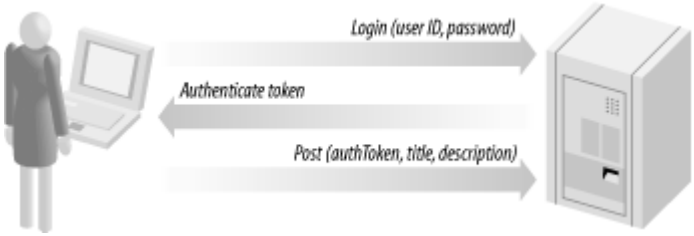
browse

Browse the database by item type. The data can be returned in either a publisher-specific XML format or as a Rich Site Summary (RSS) channel.

4.1.1 Publisher Service Security

Security in the Publisher service is handled through a login operation that returns an authorization token to the user. This token consists of a user ID, email address, login time, and a MD5 digest that the user must include in all operations that require that the user be authenticated, namely the post and remove operations (see Figure 4-1).

Figure 4-1. When registered users log in, they will be given an authentication token that they must use whenever they post or remove an item in the database



In the login operation, the user's ID and password are sent (in plain text) to the publisher service where they are validated. The service then creates an authentication token and returns it to the user. While not very secure, this illustrates one way that authentication can occur in SOAP-based web services. That is, rather than using transport-level security mechanisms, such as HTTP authentication, security can be built into the web services interface directly. In Chapter 5, we will discuss several much more secure and robust security mechanisms for web services.

4.2 The Publisher Operations

The operations exposed by the Publisher service are fairly straightforward. If we cast these operations in a Java interface, they would look like Example 4-1.

Example 4-1. The Publisher interface in Java

```
public Interface Publisher {

  public boolean register (String email,
                           String password,
                           String firstName,
                           String lastName,
                           String title,
                           String company,
                           String url);

  public boolean modify   (String email,
                           String newemail,
                           String password,
                           String firstName,
                           String lastName,
                           String title,
                           String company,
                           String url);

  public AuthInfo login   {String id,
                           String password);

  public int post         (AuthInfo authinfo,
                           String type,
                           String title,
                           String description);

  public boolean remove   (AuthInfo authinfo,
                           int itemID);
```

```
  public org.w3c.dom.Document browse (
                            String type,
                            String format,
                            int maxRows);
}
```

4.3 The Publisher Server

The Publisher Perl module uses the Perl DBI package and DBD::CSV package, both of which are available from CPAN and installed the same way SOAP::Lite is installed. The code discussed in the next section should be contained in a single Perl module called `Publisher.pm`, shown in full in Appendix C.

The code is quite straightforward. We create a database to store the news, articles, and resource items, and the list of users who will use the service. After the database is created, we define the operations for manipulating that database. Those operations are not exported. The deployed code is in the last half of the script, managing user logins and exposing the various operations that the web service will support.

4.3.1 The Preamble

Example 4-2 defines the code's namespace, loads the database module, and defines a convenience function for accessing the database handle. Data is stored in a comma-separated text file, but you can change that to a relational database by changing the `"DBI:CSV:..."` string to the data source specifier for a MySQL or a similar database.

Example 4-2. Publisher preamble

```
package Publisher;

use strict;

package Publisher::DB;

use DBI;
use vars qw($CONNECT);

$CONNECT = "DBI:CSV:f_dir=/home/book;csv_sep_char=\0";
my $dbh;

sub dbh {
  shift;
  unless ($dbh) {
    $dbh = DBI->connect(shift || $CONNECT);
    $dbh->{'RaiseError'} = 1;
  }
  return $dbh;
}
```

4.3.2 Data Tables

Example 4-3 creates the data tables for storing information about the members and items managed by the Publisher service.

Example 4-3. Create data tables

```
sub create {
  my $dbh = shift->dbh;

  $dbh->do($_) foreach split /;/, '

  CREATE TABLE members (
    memberID   integer,
    email      char(100),
    password   char(25),
    firstName  char(50),
    lastName   char(50),
    title      char(50),
    company    char(50),
    url        char(255),
    subscribed integer
  );

  CREATE TABLE items (
    itemID      integer,
    memberID    integer,
    type        integer,
    title       char(255),
    description char(512),
    postStamp   integer
  )

';

}
```

Once the tables are created, we need to write the code for manipulating the data in those tables. These methods, shown in Example 4-4, are private and will not be exposed as part of our web service. Only the first few methods are shown in full. Consult Appendix C for the full source.

Example 4-4. Methods to manipulate data in tables

```
sub insert_member {
  my $dbh = shift->dbh;
  my $newMemberID = 1 + $dbh->selectrow_array(
    "SELECT memberID FROM members ORDER BY memberID
     DESC");

  my %parameters = (@_, memberID => $newMemberID, subscribed => 0);
  my $names = join ', ', keys %parameters;
  my $placeholders = join ', ', ('?') x keys %parameters;

  $dbh->do("INSERT INTO members ($names) VALUES
           ($placeholders)", {}, values %parameters);
  return $newMemberID;
}

sub select_member {
  my $dbh = shift->dbh;
  my %parameters = @_;

  my $where = join ' AND ', map {"$_ = ?"} keys %parameters;
```

```
  $where = "WHERE $where" if $where;

  # returns row in array context and first element (memberID) in scalar
  return $dbh->selectrow_array("SELECT * FROM members
           $where", {}, values %parameters);
}

sub update_member {}

sub insert_item {}

sub select_item {}

sub select_all_items {}

sub delete_item {}
```

4.3.3 Utility Functions

Now we start defining the actual Publisher web service. Example 4-5 shows several private utility functions, primarily for dealing with the creation and validation of the authorization tokens used as part of the Publisher service's security model (discussed later).

Example 4-5. Utility functions

```
package Publisher;

use POSIX qw(strftime);

@Publisher::ISA = qw(SOAP::Server::Parameters);

use Digest::MD5 qw(md5);

my $calculateAuthInfo = sub {
  return md5(join '', 'unique (yet persistent) string', @_);
};

my $checkAuthInfo = sub {
  my $authInfo = shift;
  my   $signature   =   $calculateAuthInfo->(@{$authInfo}{qw(memberID   email
time)});
  die   "Authentication   information   is   not   valid\n"   if   $signature   ne
$authInfo->{signature};
  die "Authentication information is expired\n" if time(  ) > $authInfo-
>{time};
  return $authInfo->{memberID};
};

my $makeAuthInfo = sub {
  my($memberID, $email) = @_;
  my $time = time(  )+20*60;
  my $signature = $calculateAuthInfo->($memberID, $email, $time);
  return +{memberID => $memberID, time => $time, email => $email, signature
=> $signature};
};
```

4.3.4 Register a New User

Example 4-6 shows the code for the exported operation that registers new users.

Example 4-6. Exported method to register a new user

```
sub register {
  my $self = shift;
  my $envelope = pop;
  my %parameters = %{$envelope->method(  ) || {}};

  die "Wrong parameters: register(email, password, firstName, " .
      "lastName [, title][, company][, url])\n"
    unless 4 == map {defined} @parameters{qw(email password firstName
lastName)};

  my $email = $parameters{email};
  die "Member with email ($email) already registered\n"
    if Publisher::DB->select_member(email => $email);
  return Publisher::DB->insert_member(%parameters);
}
```

4.3.5 Modify User Information

Example 4-7 is the operation that allows users to modify their information.

Example 4-7. Exported subroutine to modify a user's information

```
sub modify {
  my $self = shift;
  my $envelope = pop;
  my %parameters = %{$envelope->method(  ) || {}};

  my $memberID = $checkAuthInfo->($envelope->valueof('//authInfo'));
  Publisher::DB->update_member($memberID, %parameters);
  return;
}
```

4.3.6 User Login

Example 4-8 is the operation that validates a user's ID and password and issues an authentication token.

Example 4-8. Exported method to validate a user and issue a token

```
sub login {
  my $self = shift;
  my %parameters = %{pop->method(  ) || {}};

  my $email = $parameters{email};
  my $memberID = Publisher::DB->select_member(email => $email, password =>
$parameters{password});
  die "Credentials are wrong\n" unless $memberID;
  return bless $makeAuthInfo->($memberID, $email) => 'authInfo';
}
```

4.3.7 Posting an Item

Example 4-9 shows the method that posts a new item to the database.

Example 4-9. Exported method to post a new item

```
my %type2code = (news => 1, article => 2, resource => 3);
my %code2type = reverse %type2code;

sub postItem {
  my $self = shift;
  my $envelope = pop;
  my $memberID = $checkAuthInfo->($envelope->valueof('//authInfo'));
  my %parameters = %{$envelope->method(  ) || {}};

  die "Wrong parameter(s): postItem(type, title, description)\n"
    unless 3 == map {defined} @parameters{qw(type title description)};

  $parameters{type} = $type2code{lc $parameters{type}}
                                    or    die    "Wrong    type    of    item
($parameters{type})\n";
  return Publisher::DB->insert_item(memberID => $memberID, %parameters);
}
```

4.3.8 Removing Items

Example 4-10 shows the exported method for removing items from the database. Only the user who added an item can remove it.

Example 4-10. Exported method to remove an item from the database

```
sub removeItem {
  my $self = shift;
  my $memberID = $checkAuthInfo->(pop->valueof('//authInfo'));
  die "Wrong parameter(s): removeItem(itemID)\n" unless @_ == 1;

  my $itemID = shift;
  die "Specified item ($itemID) can't be found or removed\n"
    unless  Publisher::DB->select_item(memberID  =>  $memberID,  itemID  =>
$itemID);
  Publisher::DB->delete_item($itemID);
  return;
}
```

4.3.9 Browsing

Users can browse the item database using either a Publisher service-specific XML format or the popular Rich Site Summary (RSS) format used extensively across the Internet.

Example 4-11, while looking fairly complex, creates the appropriate XML structures depending on the format requested by the caller.

Example 4-11. Code to support browsing in proprietary and RSS formats

```
my $browse = sub {
  my $envelope = pop;
  my %parameters = %{$envelope->method(  ) || {}};

  my  ($type,  $format,  $maxRows,  $query)  =  @parameters{qw(type  format
maxRows query)};
  $type = {all => 'all', %type2code}->{lc($type) || 'all'}
```

```
          or die "Wrong type of item ($type)\n";
  # default values
  $maxRows ||= 25;
  $format ||= 'XML';
  my $items = Publisher::DB->select_all_items($type ne 'all' ? (type =>
$type) : ( ));
  my %members;
  my @items = map {
    my ($type, $title, $description, $date, $memberID) = @$_;
    my ($email, $firstName, $lastName) = @{
       $members{$memberID} ||= [Publisher::DB->select_member(memberID =>
       $memberID)]
    }[1,3,4];
    +{
      $format =~ /^XML/ ? (
        type        => $code2type{$type},
        title       => $title,
        description => $description,
        date        => strftime("%Y-%m-%d", gmtime($date)),
        creator     => "$firstName $lastName ($email)"
      ) : (
        category    => $code2type{$type},
        title       => "$title by $firstName $lastName ($email) on "
                       . strftime("%Y-%m-%d", gmtime($date)),
        description => $description,
      )
    }
  }  @{$items}[0..(!$query  &&  $maxRows  <=  $#$items  ?  $maxRows-1  :
$#$items)];

  if ($query) {
    my $regexp = join '', map {
      /\s+and\s+/io ? '&&' : /\s+or\s+/io ? '||' : /[(  )]/ ? $_ : $_ ? '/'
      . quotemeta($_) . '/o' : ''
    } split /(\(|\)|\s+and\s+|\s+or\s+)/io, $query;
    eval "*checkfor = sub { for (\@_) { return 1 if $regexp; } return }"
         or die;
    @items = grep {checkfor(values %$_)} @items;
    splice(@items, $maxRows <= $#items ? $maxRows : $#items+1);
  }

  return $format =~ /^(XML|RSS)str$/
    ? SOAP::Serializer
        -> autotype(0)
        -> readable(1)
        ->  serialize(SOAP::Data->name(($1  eq  'XML'  ?  'itemList'  :
'channel')
                      => \SOAP::Data->name(item => @items)))
    : [@items];
};

sub browse {
  my $self = shift;
  return SOAP::Data->name(browse => $browse->(@_));
}
```

4.3.10 Search

The `search` operation is similar to the `browse` operation with the exception that users are allowed to specify a keyword filter to limit the number of items returned. It is shown in Example 4-12.

Example 4-12. Exported method to search the database

```
sub search {
  my $self = shift;
  return SOAP::Data->name(search => $browse->(@_));
}
```

4.3.11 Deploying the Publisher Service

To deploy the Publisher service, you need to do two things. First, create the database that is going to store the information. Do so by running the script in Example 4-13.

Example 4-13. Program to create the database

```
#!/usr/bin/perl -w
use Publisher;
Publisher::DB->create;
```

This will create two files in the current directory, called *members* and *items*.

Next, create the CGI script that will listen for SOAP messages and dispatch them to SOAP::Lite and the Publisher module. This is given in Example 4-14.

Example 4-14. Publisher.cgi, SOAP proxy for the Publisher module

```
#!/bin/perl -w

use SOAP::Transport::HTTP;
use Publisher;

$Publisher::DB::CONNECT =
  "DBI:CSV:f_dir=d:/book;csv_sep_char=\0";
$authinfo = 'http://www.soaplite.com/authInfo';
my $server = SOAP::Transport::HTTP::CGI
  -> dispatch_to('Publisher');
$server->serializer->maptype({authInfo => $authinfo});
$server->handle;
```

The `dispatch_to` method call instructs the SOAP::Lite package which methods to accept, and in which module those methods can be found.

Copy the CGI script to your web server's *cgi-bin* directory and install the *Publisher.pm*, *members*, and *items* files in your Perl module directory. The Publisher web service is now ready for business.

4.4 The Java Shell Client

The Java shell client is a simple interface for interacting with the Publisher web service. A typical session is shown in Example 4-15. Notice that once the shell is started, the user must log on prior to posting new items.

Example 4-15. A sample session with the Java shell client

```
C:\book>java Client http://localhost/cgi-bin/Publisher.cgi

Welcome to Publisher!
> help

Actions: register | login | post | remove | browse
> login

What is your user id: james@soap-wrc.com

What is your password: abc123xyz

Attempting to login...
james@soap-wrc.com is logged in

> post

What type of item [1 = News, 2 = Article, 3 = Resource]: 1

What is the title:
Programming Web Services with SOAP, WSDL and UDDI

What is the description:
A cool new book about Web services!

Attempting to post item...
Posted item 46

> quit

C:\book>
```

To create the shell, you need to create two Java classes: one for the shell itself (`Client.java`), and the other to keep track of the authorization token issued by the Publisher service when you log in (`AuthInfo.java`).

4.4.1 The Authentication Class

The preamble to the `authInfo` class is shown in Example 4-16.

Example 4-16. The authInfo class

```
// authInfo.java

import org.w3c.dom.Document;
import org.w3c.dom.Element;

public class authInfo {
  private int memberID;
```

```
  private long time;
  private String email;
  private byte [] signature;

  public authInfo(  ) { }

  public authInfo(int memberID, long time, String email, byte[] signature)
{
    this.memberID = memberID;
    this.time = time;
    this.email = email;
    this.signature = signature;
  }
```

The class has the usual get and set accessors. Example 4-17 shows the first four methods, and stubs the rest. For the full source, see Appendix C.

Example 4-17. authInfo accessors

```
  public void setMemberID(int memberID) {
    this.memberID = memberID;
  }

  public int getMemberID(  ) {
    return memberID;
  }

  public void setTime(long time) {
    this.time = time;
  }

  public long getTime(  ) {
    return time;
  }

  public void setEmail(String email) {}
  public String getEmail(  ) {}
  public void setSignature(byte [] signature) {}
  public byte [] getSignature(  ) {}
  public String toString(  ) {}

  public void serialize(Document doc) {
    Element authEl = doc.createElementNS(
                     "http://www.soaplite.com/authInfo",
                     "authInfo");
    authEl.setAttribute("xmlns:auth", "http://www.soaplite.com/authInfo");
    authEl.setPrefix("auth");

    Element emailEl = doc.createElement("email");
    emailEl.appendChild(doc.createTextNode(auth.getEmail(  )));

    Element signatureEl = doc.createElement("signature");
    signatureEl.setAttribute("xmlns:enc", Constants.NS_URI_SOAP_ENC);
    signatureEl.setAttribute("xsi:type", "enc:base64");
    signatureEl.appendChild(doc.createTextNode(
                            Base64.encode(auth.getSignature(  ))));

    Element memberIdEl = doc.createElement("memberID");
    memberIdEl.appendChild(doc.createTextNode(
                           String.valueOf(auth.getMemberID(  ))));
```

```
    Element timeEl = doc.createElement("time");
    timeEl.appendChild(doc.createTextNode(
                       String.valueOf(auth.getTime(  ))));

    authEl.appendChild(emailEl);
    authEl.appendChild(signatureEl);
    authEl.appendChild(memberIdEl);
    authEl.appendChild(timeEl);
    doc.appendChild(authEl);
  }
}
```

The `serialize` method creates an XML representation of the `authInfo` class instance that looks like Example 4-18.

Example 4-18. Sample serialization from the authInfo class

```
<auth:authInfo xmlns:auth="http://www.soaplite.com/authInfo">
   <email>johndoe@acme.com</email>
   <signature> <!-- Base64 encoded string --> </signature>
   <memberID>123</memberID>
   <time>2001-08-10 12:04:00 PDT (GMT + 8:00)</time>
</auth:authInfo>
```

4.4.2 The Client Class

The `Client` class is straightforward. There are utility routines for working with the SOAP client object, some code to handle authentication and login, methods to make a SOAP call for each of the operations the user might wish to perform, and then a main routine to handle the interface with the user.

4.4.2.1 Preamble

The preamble to the `Client` class is shown Example 4-19.

Example 4-19. The Client class

```
// Client.java
import java.io.*;
import java.net.*;
import java.util.*;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
import org.w3c.dom.*;

import org.apache.soap.util.xml.*;
import org.apache.soap.*;
import org.apache.soap.encoding.*;
import org.apache.soap.encoding.soapenc.*;
import org.apache.soap.rpc.*;

public class Client {

  private URL url;
  private String uri;
  private authInfo authInfo;
```

```
public Client (String url, String uri) throws Exception {
  try {
    this.uri = uri;
    this.url = new URL(url);
  } catch (Exception e) {
    throw new Exception(e.getMessage( ));
  }
}
```

The `initCall` method in Example 4-20 initializes the Apache SOAP client.

Example 4-20. The initCall method

```
private Call initCall ( ) {
  Call call = new Call( );
  call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
  call.setTargetObjectURI(uri);
  return call;
}
```

The `invokeCall` method shown in Example 4-21 makes the calls to the Publisher service. This is similar to the Hello World service example that we provided earlier.

Example 4-21. The invokeCall method

```
private Object invokeCall (Call call)
    throws Exception {
  try {
    Response response = call.invoke(url, "");
    if (!response.generatedFault( )) {
      return response.getReturnValue( ) == null
        ? null :
        response.getReturnValue().getValue( );
    } else {
      Fault f = response.getFault( );
      throw new Exception("Fault = " +
                          f.getFaultCode( ) + ", " +
                          f.getFaultString( ));
    }
  } catch (SOAPException e) {
    throw new Exception("SOAPException = " +
                        e.getFaultCode( ) + ", " +
                        e.getMessage( ));
  }
}
```

4.4.2.2 Authentication

The `makeAuthHeader` operation in Example 4-22 creates a SOAP header block that contains an authentication token. This operation must be called every time that somebody wishes to post or remove items in the Publisher service.

It works by simply creating a DOM document, instructing the `authInfo` class to serialize itself to that document (see the serialize operation on the `authInfo` class in Example 4-18), and adding the authentication information to the headers.

Example 4-22. The makeAuthHeader method

```
public Header makeAuthHeader (authInfo auth)
   throws Exception {
  if (auth == null) { throw new Exception("Oops,
    you are not logged in. Please login first"); }
  DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance(  );
  dbf.setNamespaceAware(true);
  dbf.setValidating(false);
  DocumentBuilder db = dbf.newDocumentBuilder(  );
  Document doc = db.newDocument(  );
  auth.serialize(doc);
  Vector headerEntries = new Vector(  );
  headerEntries.add(doc.getDocumentElement(  ));
  Header header = new Header(  );
  header.setHeaderEntries(headerEntries);
  return header;
}
```

4.4.2.3 User login

Example 4-23 shows the `login` operation. Notice that before we invoke the request, we must tell Apache SOAP which deserializer to use for the authentication token that will be returned if the operation is a success. The `BeanSerializer` is a utility class that comes with Apache SOAP for translating XML into instances of Java classes that conform to the Java Bean standard. We must explicitly inform Apache SOAP that we want all `authInfo` XML elements found in a SOAP message within the `http://www.soaplite.com/Publisher` namespace to be deserialized using the `BeanSerializer` class. If we don't, an error occurs whenever an `authInfo` element is found in the SOAP envelope.

We earlier brought up the topic of type mappings in Apache SOAP but never really explained what they are or how they work. A type mapping is a link between some type of native data type (such as a Java class) and the way that data type appears as XML. Serializers and deserializers are special pieces of code capable of translating between the two. The SOAPMappingRegistry is a collection of all type mappings and their corresponding serializers and deserializers.

In Apache SOAP, we have to declare a type mapping whenever we want to use any data type other than primitive built-in data types (e.g., strings, integers, floats, etc.).

Example 4-23. The login method

```
public void login (String email, String password) throws Exception {
  Call call = initCall(  );

  SOAPMappingRegistry smr =
         new SOAPMappingRegistry(  );
  BeanSerializer beanSer = new BeanSerializer(  );
  smr.mapTypes(Constants.NS_URI_SOAP_ENC,
               new QName("http://www.soaplite.com/Publisher",
                         "authInfo"),
               authInfo.class, beanSer, beanSer);

  Vector params = new Vector (  );
  params.add(new Parameter("email", String.class,
             email, null));
```

```
  params.add(new Parameter("password",
             String.class, password, null));
  call.setParams(params);
  call.setMethodName("login");
  call.setSOAPMappingRegistry(smr);
  authInfo = (authInfo) invokeCall(call);
  System.out.println(authInfo.getEmail(  ) + " logged in.");
}
```

4.4.2.4 Wrappers to call the remote operations

Although the shell client has methods for each of the operations of the Publisher web service, it doesn't necessarily have to. We've done it in this example to ensure you get a clear picture of the way the SOAP envelope gets built and used. This would be easier, though, if we had a mechanism for creating a more dynamic proxy similar to the one provided by SOAP::Lite. In Chapter 5 we will demonstrate a Java proxy built on top of Apache SOAP that does just that.

The operations in Example 4-24 all follow a very simple pattern: initialize the SOAP call, set the parameters, and invoke the SOAP call.

Example 4-24. Wrappers for the remote operations

```
public void register (String email,
                      String password,
                      String firstName,
                      String lastName,
                      String title,
                      String company,
                      String url) throws Exception {
  Call call = initCall(  );

  Vector params = new Vector (  );
  params.add(new Parameter("email", String.class, email, null));
  params.add(new Parameter("password", String.class, password, null));
  params.add(new Parameter("firstName", String.class, firstName, null));
  params.add(new Parameter("lastName", String.class, lastName, null));
  if (url != null)
             params.add(new Parameter("url", String.class, url, null));
  if (title != null)
             params.add(new Parameter("title", String.class, title, null));
  if (company != null)
            params.add(new   Parameter("company",   String.class,   company,
null));
  call.setParams(params);
  call.setMethodName("register");
  invokeCall(call);
  System.out.println("Registered.");
}

public void postItem (String type,
                      String title,
                      String description)
                      throws Exception {
  Call call = initCall(  );
  Vector params = new Vector (  );
  params.add(new Parameter("type", String.class, type, null));
  params.add(new Parameter("title", String.class, title, null));
  params.add(new    Parameter("description",    String.class,    description,
null));
```

```
  call.setParams(params);
  call.setMethodName("postItem");
  call.setHeader(makeAuthHeader(authInfo));
  Integer itemID = (Integer)invokeCall(call);
  System.out.println("Posted item " + itemID + ".");
}

public void removeItem (Integer itemID);
public void browse (String type,
                    String format,
                    Integer maxRows);
```

4.4.2.5 The main routine

Now that the basic operations for interacting with the web service have been defined, we need to create the code for the Publisher shell (Example 4-25). This code does nothing more than provide users with a menu of things that can be done with the Publisher service. In a loop we get input from the user, decide what they want to do, and do it.

Because none of this code deals directly with the invocation and use of the Publisher web service, significant pieces were removed for the sake of brevity. The entire code sample can be found in Appendix C.

Example 4-25. The main method

```
  public static void main(String[] args) {
    String myname = Client.class.getName(  );

    if (args.length < 1) {
      System.err.println("Usage:\n  java " + myname + " SOAP-router-URL");
      System.exit (1);
    }

    try {
      Client          client          =          new          Client(args[0],
"http://www.soaplite.com/Publisher");

      InputStream in = System.in;
      InputStreamReader isr = new
            InputStreamReader(in);
      BufferedReader br = new BufferedReader(isr);
      String action = null;
      while (!("quit".equals(action))) {
         System.out.print("> ");
         action = br.readLine(  );

         if ("register".equals(action)) {
             // code hidden for brevity
             client.register(email, password, firstName, lastName,
                             title, company, url);
         }

         if ("login".equals(action)) {
             // code hidden for brevity
             client.login(id,pwd);
         }
```

```
            if ("post".equals(action)) {
                // code hidden for brevity
                client.postItem(type, title, desc);
            }

            if ("remove".equals(action)) {
                  // code hidden for brevity
                client.removeItem(Integer.valueOf(id));
                } catch (Exception ex) {
                   System.out.println("\nCould not remove item!");
                }
                System.out.println(  );
            }

            if ("browse".equals(action)) {
                  // code hidden for brevity
               client.browse(type, format, ival);
                } catch (Exception ex) {
                   System.out.println(ex);
                   System.out.println("\nCould not browse!");
                }
            }

            if ("help".equals(action)) {
                System.out.println("\nActions:  register  |  login  |  post  |
remove | browse");
            }
        }
      } catch (Exception e) {
        System.err.println("Caught Exception: " + e.getMessage(  ));
      }
    }
}
```

4.4.3 Deploying the Client

Once the code is written, compile it and launch it with the following command:

```
C:\book>java Client http://localhost/cgi-bin/Publisher.cgi
```

Replace `localhost` with the name of the web server where the Publisher CGI script is deployed. Figure 4-2 shows the shell in action.

Figure 4-2. The Publisher shell at runtime